Optimizing Active Database Transactions Using an Extended Multiversion Concurrency Control Protocol

Dimitri Tombroff, François Llirbat, Eric Simon

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Optimizing Active Database Transactions Using an Extended Multiversion Concurrency Control Protocol

Dimitri Tombroff, Francois Llirbat, Eric Simon

N 2519

Mars 1995

PROGRAMME 1

ISSN 0249-6399

Optimizing Active Database Transactions Using an Extended Multiversion Concurrency Control Protocol

Dimitri Tombroff*, Francois Llirbat*, Eric Simon*

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet Rodin

Rapport de recherche n°2519 — Mars 1995 — 24 pages

Abstract: We study the problem of efficiently evaluating transactions that automatically invoke the execution of (deferred) database triggers at the end of the transaction. In particular, we consider an important class of triggers which may express arbitrary integrity constraints and alerters. Their event part specifies data modifications, their condition part is an arbitrary database query, and their action part can raise some alerts, issue a rollback, or "repair" the data modification that triggered the rule. An update transaction that invokes such deferred trigger(s) reads (and locks) new database items before committing. These read operations may entail inter-transactions blockings, thereby degrading the performance of active database applications. We propose a slight extension of the classical multiversion two phase locking (MV2PL) protocol whereby these reads access versions and do not take locks. We prove the correctness of this protocol, and show that its implementation requires very few changes to classical implementations of MV2PL. Finally, a careful performance evaluation conducted with a simulator, shows the benefits of our protocol compared to a two phase locking protocol.

Key-words: Active database systems, database triggers, concurrency control algorithms.

(Résumé : tsvp)

Submited to the 1995 VLDB Int. Conf.

*{Dimitri.Tombroff}{Francois.Llirbat}{Eric.Simon}@inria.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Té lé phone : (33 1) 39 63 55 11 —Té lé copie : (33 1) 39 63 53 30

Un Contrôle de Concurrence Multiversion pour Optimiser les Performances des Transactions dans un SGBD Actif

Résumé : Dans les bases de données actives, les transactions peuvent déclencher et exécuter des triggers. Dans cet article, nous considérons une classe importante de triggers qui permettent d'exprimer des contraintes d'intégrités générales. Ces triggers sont déclenchés par les modifications des données, leur condition est une requête arbitraire sur la base, et leur action peut déclencher un alerteur, entraîner l'abandon de la transaction ou compenser certaines des modifications de la transaction qui ont déclenché le trigger. Une transaction qui déclenche de tels triggers en mode différé lit et verrouille des données supplémentaires avant de valider (ou d'être abandonnée). Ceci peut causer des bloquages entre transactions et dégrader les performances. Nous proposons une légère extension du contrôle de concurrence multiversion classique (MV2PL) qui permet à ces lectures d'accéder des versions sans verrouiller les données. Nous prouvons la correction de notre algorithme et nous montrons qu'il peut être realisé facilement à partir d'un MV2PL existant. Finalement, une étude par simulation montre les apports de notre algorithme par rapport à un protocole de verrouillage à deux-phases.

Mots-clé : Bases de données actives, triggers, algorithmes de contrôle de concurrence.

1 Introduction

In relational active database systems (see [WCD95] for a broad overview and a description of systems), database triggers are rules[1] with: an event that causes the rule to be triggered, a condition that is checked when the rule is triggered, and an action that is executed when the rule is triggered and its condition is true. Typical actions are database modifications, procedures that raise some alerts, or a rollback statement that aborts the transaction. Events are issued by transactions and generally consist of database statements such as data modifications, data retrievals, or transactional commands.

At specific points in a transaction's execution the database system takes a set of events issued by the transaction, automatically retrieves the triggered rules, and processes them. There are two kinds of rule processing points: rules can be triggered immediately after (or before) each occurrence of an event in the transaction (*immediate* execution mode), or at the end of the transaction (*deferred* execution mode). Triggers are defined as immediate or deferred and this determines subsequently their execution mode in a transaction. In most active database systems (and at least, in all commercial active database systems), the execution of triggers is done within the triggering transaction. Thus, all database read and write operations that are required for the execution of triggers are viewed by the database system's transaction manager as regular operations of the triggering transaction.

In this paper, we assume that all triggers are deferred, and have an action part that can raise an alert, generate a rollback, or overwrite database items that have already been modified by the triggering transaction. These triggers, henceforth called RCA[2] triggers, may represent general integrity constraints that either issue a rollback or "repair" an update when a constraint is violated. Thus, RCA triggers play a key role in the design of many applications (e.g., accounting/OLTP applications). The execution of an update transaction that triggers RCA rules has two distinct parts. First, it executes the database statements specified in the transaction's text which amounts to perform read and write operations on the database. Then, at the end of the transaction, it executes a set of RCA triggers, which amounts to performing read operations on arbitrary database items, and write operations to database items already written in the first part[3].

[1] We shall indifferently use the words trigger or rule

[2] RCA stands for Rollback, Compensative, Alerter triggers

[3] As explained in [SWY93], a rollback statement can be viewed as a write operation

With classical strict two phase locking (S2PL), or multiversion two phase locking (MV2PL) protocols, the database items read during the second part (i.e., the trigger part) of an update transaction T are not accessible by another concurrent transaction T' that wants to write them. Thus, T' will be waiting for T's release of its locks. In particular, suppose that T updates a relation R_1 and triggers an RCA rule that needs to read a relation R_2. Then concurrent transactions that update R_2 will conflict with T. In this scenario, relation R_2 may become a source of lock contention in the application, and thereby reduce the transaction throughput of the application. In mission-critical applications such as OLTP, this behavior is unaffordable.

The major contribution of this paper is to propose an extension of the MV2PL protocol, called EMV2PL protocol, in which read operations, resulting from the execution of RCA triggers in an update transaction, are allowed to access versions of database items exactly as read-only transactions do in the MV2PL Protocol. Thus, read operations for RCA triggers do not acquire locks on the database and do not impede the transactional traffic in the application. We formally prove the correctness of our new protocol. We show that this protocol can be efficiently implemented with a very few modifications to classical implementations of the MV2PL protocol. Finally, we present careful simulation results that show the value of our protocol compared to the S2PL protocol.

The remaining of the paper is structured as follows. Section 2 presents our active database framework. Section 3 describes the optimization problem and gives the principles of our solution. Section 4 presents our extended multiversion two phase locking protocol. Section 5 discusses implementation issues. Simulation results are presented in Section 6. Section 7 compares our work with related work. Finally, Section 8 concludes.

2 Database Triggers and Transactions

In this section, we present the active database framework considered throughout this paper. We shall assume that the reader is familiar with basic concepts of transaction management ([Ull88], Chap. 9).

In this paper, database triggers consist of three parts: an event that causes the rule to be triggered, a condition that is checked when the rule is triggered, and an action that is executed when the rule is triggered and its condition is true. Events are restricted to data modifications (insert, delete, or update) on a particular relation, conditions are arbitrary queries over the database, and actions may raise some alerts and execute data modification, retrieval, or rollback operations. We assume that all triggers are defined as *deferred* triggers, which means that they are executed at the end of the transaction and within the transaction[4]. This way, a transaction execution has two consecutive parts: first, the statements specified in the transaction program's text are processed, and then triggers are processed. For convenience, the first part

[4] For instance, in SQL3, it is possible to execute transactions with a special cursor mode called CASCADE OFF, in which all triggers are deferred until commit time

is called the *program part* and the second part is called the *trigger part*. Finally, we assume that transactions are executed in total isolation (corresponding to SQL degree 3 isolation [GR93]).

We restrict ourselves to a class of triggers, called RCA triggers, whose action part can raise an alert, generate a rollback, or overwrite database items that have already been modified in the program part of the triggering transaction. Before formally defining this class of triggers, a few definitions will be useful.

Let us suppose that two relations, ins_R and del_R, are associated with every relation R. They respectively contain the tuples that have been inserted, and deleted between the beginning of the transaction and the current state. Thus, if I_k is the current database state reached by a transaction, and I_0 is the initial database state, for any relation R, we have:

$$\begin{array}{lcl} I_k[R] - I_0[R] & = & I_k[ins\_R] \\ I_0[R] - I_k[R] & = & I_k[del\_R] \\ I_k[ins\_R] \cap I_k[del\_R] & = & \emptyset \end{array}$$

where $I_k[R]$ denotes the instance of relation R in state I_k.

RCA trigger: Let r be a trigger defined on a relation R. If for any state I_k in which r is executed, every operation in the action of r can neither

- increase the cardinality of $I_k[ins_R]$ and $I_k[del_R]$, nor
- modify any instance of a relation other than R,

then r is said to be an *RCA trigger*.

Depending on the particular trigger language used, it is easy to derive syntactic sufficient conditions that determine if a trigger is an RCA trigger.

Example 1: Suppose we have two relations *Purchase* (customer, item, quantity, supplier) and *Supplier* (supplier, address). A simple constraint forbids a transaction from deleting a supplier if there exists some item purchased from this supplier. The following trigger enforces this constraint by undoing deletions to *Supplier* that violate the constraint. Using a notation inspired from SQL3[5] ([Mel93]), this trigger could be written as:

```
after delete on Supplier
then  insert into Supplier
      select (s,a) from deleted
      where exists (select * from Purchase
                    where supplier = s );
```

In this trigger, the relation *deleted* refers to *del_Supplier*.

[5] The concrete syntax for triggers is irrelevant in this paper; it is just used for illustration

3 Problem Statement and Solution Overview

Suppose we have two relations: *Account* (account-id, name, balance, ...), that stores bank accounts, and *Withdraw* (account-id, date, ...), that keeps track of withdraws on bank accounts. We have two transaction programs *purchase* and *debit*. The *purchase* transaction inserts a tuple into *Withdraw* when an item is purchased by a customer owning a given bank account. The *debit* transaction reads relation *Withdraw* from a particular date and updates bank account's balances accordingly. Finally, we have defined an RCA trigger that says that when an insert to *Withdraw* occurs, if the amount of the withdraw is greater than the balance of the corresponding bank account, then the transaction must be rolled back.

Suppose that transactions run in isolation degree 3 and obey the strict two phase locking policy [BHG87]. Whenever *purchase* executes, the RCA rule is triggered and its condition is evaluated, which amounts to read relation *Account*. Thus, *purchase* needs to obtain a write lock on *Withdraw* and then a read lock on *Account*. Executions of *purchase* may be conflicting with concurrent executions of *debit* since they respectively want to read and write relation *Account*. When a conflict occurs between two transactions, one transaction is blocked and waits that the other one releases its locks (when committing or aborting). Thus, running instances of *purchase* augment the lock contention on *Account* and impede the transactional traffic in the database system.

It is worthwhile noticing that an update transaction that invokes RCA triggers has a peculiar property regarding locking. During the program part of the transaction, read and write locks are acquired, whereas during the trigger part, only *new* read locks are acquired. This property is due to the restriction of RCA triggers that can only overwrite database items that were previously written by the transaction. For this reason, we shall call such transactions *Write-then-Read* transactions.

Our basic optimization idea allows the trigger part of an update transaction to read versions of database items exactly as read-only transactions do in the MV2PL protocol[6]. In this protocol, transactions are either *read-only* transactions or *update* transactions. Each read-only transaction T is assigned a startup timestamp $sn(T)$ when it begins to execute, and each update transaction T is assigned a timestamp $tn(T)$ when it commits. Update transactions are serialized together through the usual strict two phase locking protocol. Instead of simply updating the data items, update transactions create new *versions* of the data items. These versions are timestamped with the timestamp tn of their creator. Hence, several ordered versions may exist for a single data item. When a read only transaction whishes to read an item, it simply reads the most recent version having a timestamp smaller than its startup timestamp. Thus, a read-only transaction reads only versions created by update transactions that committed before it started. The major advantage of this protocol is that read-only transactions do not acquire locks.

[6] In [BHG87] it is called *Multiversion Mixed Method*. Multiversion two phase locking is used for a different protocol

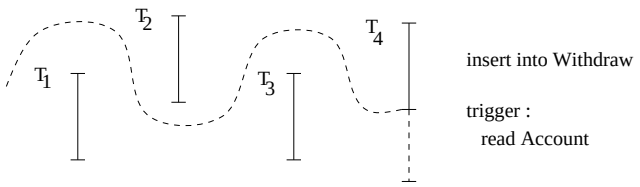

Figure 1: T_4 is serialized after T_2 and before T_1 and T_3.

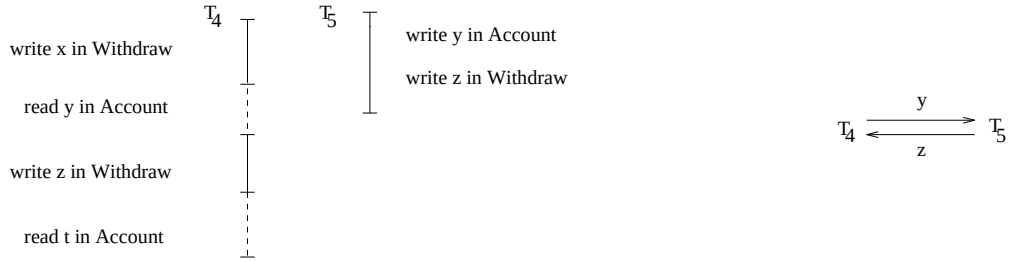

Figure 2: T_4 does not see T_5's modification on y but sees T_5's modification on z.

Coming back to our previous example, suppose that three instances of *debit*, T_1, T_2, and T_3, execute concurrently with an instance of *purchase*, noted T_4. The concurrent execution of these transactions is shown on Figure 1. Let us apply the principles of MV2PL to the trigger part of T_4. Relation *Account* is not locked by T_4 since T_4 will read a version of *Account* that corresponds to the snapshot represented by the dotted line in Figure 1. Thus, the possible conflicts on *Account* may only occur between T_1, T_2, and T_3, and T_4 is not capable of blocking these transactions. Furthermore, the above execution is correct as far as serialization is concerned. Transaction T_2, which commits before the trigger part of T_4 starts is serialized before T_4. Transactions T_1 and T_3, which commit after the trigger part of T_4 starts are serialized after T_4 because they commit after T_4 acquired all its locks.

However, the usage of MV2PL to monitor the trigger parts of update transactions can raise some problems. First, note the importance of our assumption that all triggers are deferred. Suppose that T_4 is changed to perform several inserts into $Withdraw$, and the trigger is defined as immediate. Then, the trigger and the program parts of the transaction are interleaved. This means that a new item could be write-locked after the first trigger part of T_4, and a serialization fault might occur if a concurrent transaction, say T_5, executes and writes both $Withdraw$ and *Account*, as shown on Figure 2. The edges of the serialization graph[7], labelled with the item that causes a conflict, are shown in this figure. Thus, although the semantics of

[7] This graph representing the serialization order of transactions is defined in Section 4

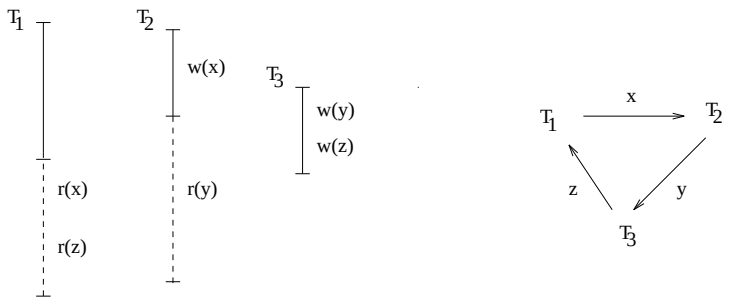

Figure 3: Concurrent execution of Write-then-Read Transactions.

the transaction would not be changed by defining the trigger as immediate, our optimization opportunity would be lost.

Even if triggers are deferred, the main problem still occur when concurrent Write-then-Read transactions are executed. Suppose we have three concurrent executions of update transactions, T_1, T_2, and T_3, as depicted in Figure 3. T_1 reads a version of x created before T_2 since $T2$ commits after the trigger part of T_1 starts. Using the same argument, T_2 reads a version of y created before T_3. Finally, T_1 reads a version of z produced by T_3. As the figure shows, there is a cycle in the serialization graph and execution is not correct.

We will show that whenever such a problem occurs, there is a "critical" read(x) operation issued by the trigger part of a transaction T that occurs after a write(x) operation issued by a transaction T', and the trigger part of T starts while the trigger part of T' is running. We therefore propose a protocol that dynamically prevents such "critical" reads from happening.

4 Extended Multiversion Two Phase Lock Protocol

4.1 Preliminaries

We first recall notations, definitions, and results from [BHG87] and [AS89].

A database is a set of objects. A Transaction T_i is an ordered pair $(\Sigma_i, <_i)$ where Σ_i is the set of operations in T_i and $<_i$ is the execution order of these operations. Read or write operations executed by T_i are noted $r_i[x]$ or $w_i[x]$, respectively. Transactions terminate either by a commit (c_i for T_i) or an abort (a_i for T_i). Let $T = \{T_1, T_2, ...T_n\}$ be a set of transactions. An *history* H of T is a partial order $(\Sigma, <_H)$ where Σ is the set of operations executed by transactions in T, and $<_H$ is the execution order of those operations.

Two histories H_1 and H_2 are *conflict equivalent* if (i) they are defined over the same set of transactions and (ii) if o_1 and o_2 are two conflicting operations and $o_1 <_{H_1} o_2$ then $o_1 <_{H_2} o_2$. An history H_s is *serial* if for every two transactions

T_i and T_j either all operations of T_i precede all operations of T_j or vice-versa. An history H is *serializable* if it is equivalent to a serial history. One determines if H is serializable by analyzing the *Serialization Graph* of H , noted SG(H). This is a directed graph where nodes are the committed transactions in H, and there is an edge $T_i \rightarrow T_j$ if one of T_i's operation precedes and conflicts with one of T_j's operation. An history H is serializable if its serialization graph is acyclic.

In a multiversion database, each write operation of an object x produces a new *version* of x. Thus for each object x, there is a list of versions written $x_i, x_j, ..$ where the subscript is the index of the transaction that wrote the version (thus, x_i represents the version of x created by T_i). A *multiversion history* (MV) is the sequence of operations on the versions of objects submitted by transactions. Each write operation $w_i[x]$ is mapped into $w_i[x_i]$ and each operation $r_i[x]$ is mapped into $r_i[x_k]$, for some k. A transaction T_j *reads-x-from* T_i in H if $r_j[x_i] \in H$. Notice the only conflicting operations in H are $w_i[x_i]$ and $r_j[x_i]$ for some x, T_i and T_j. Two MV histories are equivalent if they have the same operations. An MV history is *one-copy serializable* if it is equivalent to a serial history over the same set of transactions executed over a single version database. The *multiversion serialization graph* of an MV history H (MVSG(H)) is a directed graph whose nodes represent committed transactions. There is an edge $T_i \rightarrow T_j$ in MVSG(H) if

(i) $r_j[x_i] \in H$ for some x (that is, T_j reads-x-from T_i)

Such an edge is called an SG(H) edge in [BHG87]. Additional edges are defined as follows. For each object x, there is a total order (noted $\ll_x$) on all transactions that write x. One adds the edge $T_i \rightarrow T_j$ to MVSG(H) iff one of the following holds:

(ii) for some x and T_k, T_i reads-x-from T_k and $x_k \ll_x x_j$
(iii) for some x and T_k, T_k reads-x-from T_j and $x_i \ll_x x_j$

These additional edges are called *version order edges*. An MV history is one-copy serializable if its multiversion serialization graph is acyclic ([BHG87]).

4.2 The EMV2PL Protocol

We now present our Extended MV2PL protocol, called EMV2PL. In this protocol, we distinguish *read-only* transactions, noted R transactions, from update transactions. Moreover, it will be convenient to subdivide update transactions into transactions that only have a program part (i.e., no rule is triggered) and transactions that have both a program part and a trigger part. The former are called Write transaction (noted W transactions) whereas the later are called Write-then-Read transactions (noted W|R transactions). Figures 4 and 5 respectively show how operations issued by R and W transactions are processed by the Transaction Manager (TM). An R transaction first obtains a *start number* noted *sn* from TM. Then every read(x) gets the most recent version of x having a timestamp less than or equal to

Operation Invocation	Operation Execution
$begin(T)$	get $sn(T)$ from TM
...	
$read(x)$	return x's version with largest version number $\leq sn(T)$
...	
$end(T)$	ϕ

Figure 4: Execution of an R transaction

Operation Invocation	Operation Execution
$begin(T)$	ϕ
...	
$read(x)$	get read-lock on x /*may wait according to the 2PL protocol*/ return the most recent version of x
...	
$write(y)$	get write-lock on y /*may wait according to the 2PL protocol*/ creates a new version of y
...	
$end(T)$	get $tn(T)$ from TM $commit(T)$: perform database updates with version number $tn(T)$ release locks

Figure 5: Execution of a W transaction

sn. Reads in a W transaction follow the usual S2PL protocol, whereas a write(x) creates a new version of x (if x is written for the first time). Before committing, a W transaction obtains its *transaction number* (noted tn), associates this number to each of its versions, and releases all its locks.

Figure 6 shows how the operations issued by a W|R transactions are processed. The program part of the transaction is processed as a W transaction. When the end of the program part is reached, the transaction signals the beginning of the trigger part to the TM and receives a transaction number tn. After that point, read and write operations are processed as follows. A read(x) operation invokes a function *check_read*(x) that checks if there is an uncommitted version[8] of x created by another transaction whose number is smaller than the caller's tn. In that case, *check_read* waits until that transaction commits. After that, the W|R transaction reads the most recent version of x with timestamp smaller than or equal to tn. A write(x) operation only modifies a version already created in the program part.

[8] i.e., a version created by a still active transaction

Operation Invocation	Operation Execution
$begin(T)$	ϕ
...	
$read(x)$	get read lock on x /*may wait according to the 2PL protocol*/ return the most recent version of x
...	
$write(y)$	get write-lock on y /*may wait according to the 2PL protocol*/ creates a new version of y
...	
$begin\_trigger(T)$	get $tn(T)$ from TM
...	
$read(z)$	$check\_read(z)$ /*may wait (see EMV2PL protocol)*/ return z's version with largest version number $\leq tn(T)$
...	
$write(t)$	update the last version of t /*this version was created by T before $begin\_trigger(T)$ */
...	
$end(T)$	$commit(T)$: perform database updates with version number $tn(T)$ release locks

Figure 6: Execution of W|R transaction

Before committing, a W|R transaction associates its tn to each version it created and releases all its locks.

To maintain the tn's, the TM uses a monotonically increasing counter. Since W and W|R transactions obtain their tn after they acquired their last locks and before committing, tn's are *lock-points*[9]. For R transactions, the TM simply guarantees that their sn is smaller than the tn of any active or forthcoming transaction. Thus, an R transaction reads only versions of committed transactions.

4.3 Correctness

In this section, we prove that the EMV2PL protocol guarantees serializability of all transactions. The Lemma states that R and W|R transactions do not execute "critical" reads. This lemma is then used to prove that the EMV2PL protocol is one-copy serializable.

For the ease of presentation, we denote $sn(r_i[x_k])$ the timestamp used by T_i to select version x_k. If r_i is executed in an R transaction then $sn(r_i[x]) = sn(T_i)$. If r_i is executed in the trigger part of a W|R transaction then $sn(r_i[x]) = tn(T_i)$. For the purpose of uniformity, we consider as in [AS89] that $sn(r_i[x]) = \infty$ for any other

[9] A lock point of a transaction is any point in time between the last lock acquired and the first lock released

read operations since these reads use locks and always access the latest version of an item.

Lemma : If $w_j[x_j] <_H r_i[x_k]$ $(i \neq j)$ with $x_k \ll_x x_j$ then $sn(r_i[x_k]) < tn(T_j)$.

Proof : First observe that if $w_j[x_j] <_H r_i[x_k]$ then $r_i[x_k]$ was executed either in an R transaction or in the trigger part of a W|R transaction. Otherwise, T_i would have taken a shared lock on x and would have read x_j.

If $w_j[x_j] <_H c_j <_H r_i[x_k]$ then $sn(r_i[x_k]) < tn(T_j)$. Otherwise T_i would have read x_j. If $w_j[x_j] <_H r_i[x_k] <_H c_j$ and if T_i is an R transaction, then T_j was active (or not yet started) when T_i obtained its timestamp $sn(T_i)$. Thus, we have $sn(r_i[x_k]) = sn(T_i) < tn(T_j)$. If T_i is a W|R transaction, $r_i[x_k]$ was not delayed after c_j by the *check_read* function. If *check_read* was called after $w_j[x_j]$ then $sn(r_i[x_k]) = tn(T_i) < tn(T_j)$. If *check_read* was called before $w_j[x_j]$[10] then T_j had not reached yet its lock point while T_i did, and $sn(r_i[x_k]) = tn(T_i) < tn(T_j)$ □.

Theorem : The EMV2PL protocol guarantees serializability of all transactions.

Proof : Since only strict histories are accepted, we consider only committed transactions. Given an history H produced by the EMV2PL protocol, we prove that MVSG(H) is acyclic. Let $ts(T_i)$ be defined as follows:

$$ts(T_i) = \begin{cases} sn(T_i) & \text{if } T_i \text{ is an R transaction} \\ tn(T_i) & \text{otherwise} \end{cases}$$

We show that MVSG(H) is acyclic by showing that for each of its edges $T_i \rightarrow T_j$, $(i \neq j)$, $ts(T_i) < ts(T_j)$. We perform a case analysis on the types of edges.

(i) $T_i \rightarrow T_j$ is an SG(H) edge. That is, $w_i[x_i] <_H c_i <_H r_j[x_i]$. We thus have $tn(T_j) < sn(r_j[x_k])$. If $sn(r_j[x_k])$ is finite, then $tn(T_j) < ts(T_i)$ (see definition of $sn(r_j[x_k])$). If $sn(r_j[x_k]) = \infty$ then T_j acquired a shared-lock on x after T_i released an exclusive-lock. Since $tn(T_i)$ and $tn(T_j)$ are lockpoints, we have $tn(T_i) < tn(T_j)$ (recall that with the S2PL policy, if $T_i \rightarrow T_j$, *all* lock points of T_i precede *all* lock points of T_j). Thus in both cases, $ts(T_i) < ts(T_j)$.

(ii) $T_i \rightarrow T_j$ is a version order edge because for some x and T_k, $r_i[x_k] \in H$ and $x_k \ll_x x_j$. If $sn(r_i[x_k]) = \infty$ then T_i acquired a shared-lock on x. Also T_i acquired its lock before T_j acquired an exclusive-lock on x (otherwise, T_i would have read x_j instead of x_k). We have then $ts(T_i) < ts(T_j)$ since $ts(T_i)$ and $ts(T_j)$ are lock points of T_i, T_j respectively. If $sn(r_i[x_k])$ is finite then suppose that $r_i[x_k] <_H w_j[x_j]$, then $ts(T_i)$ was obtained before $ts(T_j)$ and it follows that $ts(T_i) < ts(T_j)$. Suppose now that $w_j[x_j] <_H r_i[x_k]$. From the previous Lemma, we have $ts(T_i) = sn(r_i[x_k]) < tn(T_j)$.

(iii) $T_i \rightarrow T_j$ is a version order edge because for some x and T_k, $r_k[x_j] \in H$ and $x_i \ll_x x_j$. Since $x_i \ll_x x_j$, T_i acquired an exclusive-lock on x before T_j, so we have $ts(T_i) < ts(T_j)$. □

[10] We suppose that $w_j[x_j]$ is the first $w_j[x_j]$ of T_j since if $w_j[x_j] <_H r_i[x_k]$ holds for any $w_j[x_j]$ of T_j, it holds for the first one

4.4 Deadlocks

Clearly, EMV2PL suffers from deadlocks since it uses S2PL for serializing W transactions and the program parts of W|R transactions. However, once a W|R transaction starts executing its trigger part, it may not be involved anymore in deadlocks. This is easily shown as follows.

Suppose T_i is a W|R transaction which executes its trigger part. It thus has already obtained its tn. A deadlock implies a cycle in the transaction wait-for graph[11]. T_i is involved in a deadlock if it belongs to a cycle or if it is waiting for a transaction from a cycle. We first show that T_i may not belong to a cycle. Let $T_i \rightarrow T_{i_1} \rightarrow T_{i_2} \rightarrow \ ... \ T_{i_k} \rightarrow T_i$ be a cycle. Since T_i is waiting for T_{i_1}, then T_{i_1} has also obtained its tn and $tn(T_i) > tn(T_{i1})$ (only in this case could *check_read* have blocked T_i). Furthermore, T_{i1} may not be a W transaction because once a W transaction has obtained its tn (i.e., has passed its lockpoint), it may not be waiting for another transaction T_{i_2}. Thus, T_{i_1} is a W|R transaction which executes its trigger part. Applying this for every node of the cycle, we obtain that all the nodes are W|R transactions and $tn(T_i) > tn(T_i)$, which is a contradiction.

Suppose now that T_i does not belong to a cycle. If there was a path from T_i to a transaction of a cycle, say T_j, then T_j would also be a W|R transaction executing its trigger part. This is impossible since it belongs to a cycle □.

This result is significant in that it shows that EMV2PL prevents RCA-triggers from causing deadlocks.

5 Implementation Issues

The EMV2PL protocol may use the same mechanisms as MV2PL to maintain timestamps and versions. Therefore, we mainly discuss here the implementation of the *check_read(x)* function. All the informations needed by *check_read*(x) are found in the lock table. Indeed, checking if there is an uncommitted version of x only requires to check if there is a X-lock taken on x. If *check_read*(x) must wait for x's creator to commit, it simply waits for the release of this lock. We show that these mechanisms are easily implemented on top of an existing MV2PL lock manager.

We take a lock manager similar to the one described in [GR93]. For simplicity we only consider exclusive locks (X) and shared locks (S). The lock manager's data structures are summarized in Figure 7. Its two basic interfaces are *lock* and *unlock*. The only change in the lock manager's data structures consists of associating to each lock header a list of process ids, referred to as *list_pid*. This list is used to store the pid(s) of W|R transactions blocked by *check_read*().

Given this, *check_read*(x) performs the following. It looks at the lock header associated with x and checks if x is X-locked. If yes, there is an uncommitted version of x and *check_read*() checks if the owner of this lock has a tn smaller than the caller's

[11] Nodes of the wait-for graph are transactions and there is an edge $T_i \rightarrow T_j$ iff T_i is blocked by T_j (see [BHG87])

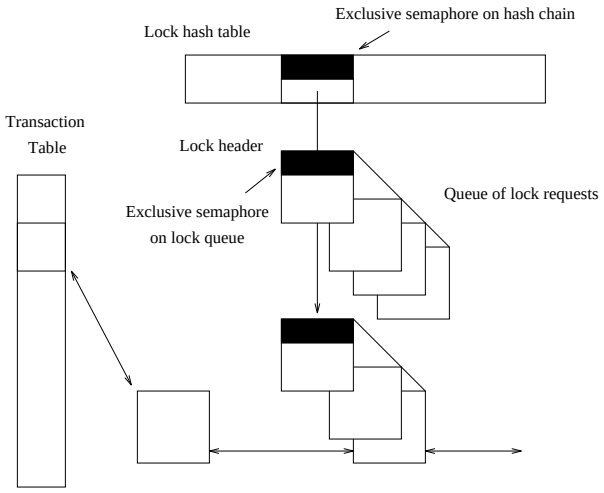

Figure 7: Lock manager data structures

one. In that case, *check_read*() appends the caller's pid into the lock header *list_pid* and waits for the release of the X-lock. After it woke up, or if it was not blocked, *check_read* returns an ok message. These operations require neither to traverse the lock request queue nor to insert a new lock request into the lock request queue.

Finally, we slightly modify the *unlock* interface so that a transaction releasing an X-lock wakes up all processes whose pids are in the lock header *list_pid* and set *list_pid* to null. This is the only modification of the unlock routine.

Notice the *lock* interface is unchanged. In fact, *check_read* has less overhead than *lock*[12] in terms of lock queue traversal and memory space, since instead of enqueuing new lock requests, it (sometimes) appends the caller's pid into a list. The number of locks in the system is thus reduced. We give in Appendix 1, the pseudo-code for *check_read*, assuming implementation context described in [GR93].

6 Performance Evaluation

The EMV2PL protocol exposes to a clear tradeoff between the increased concurrency and the I/O and storage overhead caused by the use of versions. This tradeoff was studied in detail in [CM86] and [BC92]. Our EMV2PL protocol enables the use of versions within update transactions. Thus we are led to examine how EMV2PL behaves with respect to this tradeoff. We developed a simulation model highly inspired from [Car83], [CM86] and [ACL87]. This model was implemented using the SIM package [ADW92].

[12] which would be invoked instead of *check_read* in MV2PL or 2PL

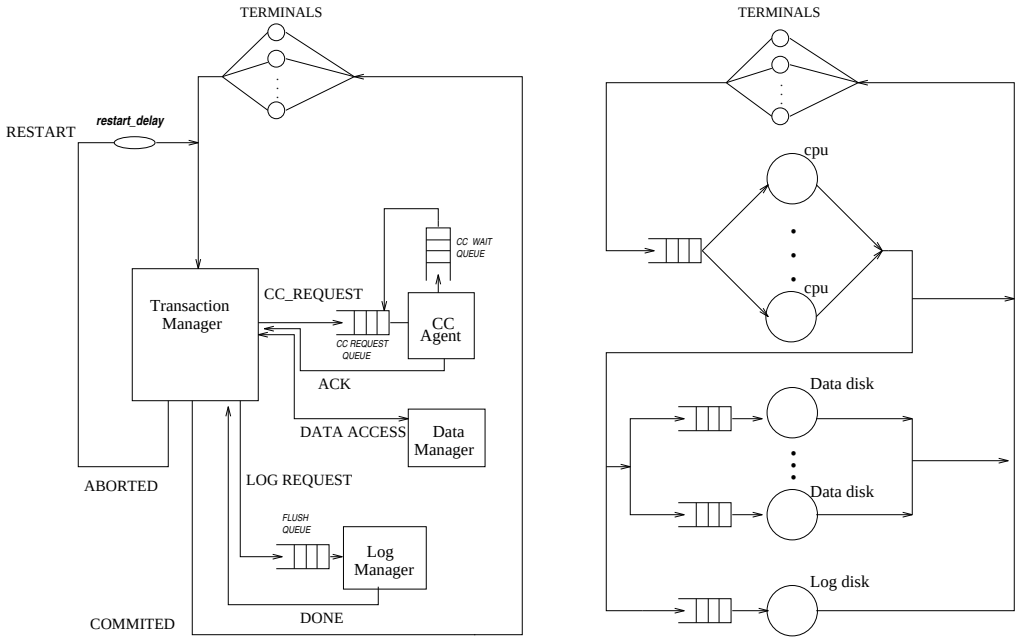

Figure 8: The Simulation Model

6.1 The Simulation Model

In our simulation, we model the concurrent execution of transactions on a single site database.

The workload parameters are the following. Parameter WR_frac is the fraction of terminals executing W|R transactions. Parameter W_size is the average number of read-write operations executed by the W transaction and by the program part of W|R transactions. It is the mean of a uniform distribution between $W\_size \pm 2$. Parameter R_size represents the number of read operations executed by the W|R transactions in their trigger parts. We assume that triggers only perform reads and access objects in a sequential manner. These parameters and their values for the experiments are summarized in Table 1.

The database simulation model (Figure 8a) is divided into four main components: a Transaction Manager (TM), a Concurrency Control Agent (CCA), a Data Manager (DM) and a Log Manager (LM). The TM is responsible for issuing concurrency control requests and their corresponding database operations. It also provides the durability property by flushing all log records of committed transactions to durable memory. The CCA schedules the concurrency control requests according to the S2PL or EMV2PL protocol. The LM provides read and insert-flush interfaces to the log table. The DM is responsible for granting access to the physical data objects and executing the database operations.

Parameter	Description	Value
db_size	Number of objects in database	3000 objects
num_terms	Number of terminals	25
WR_frac	fraction of W|R transactions	10% to 100%
W_size	mean size of program part	5 pages
R_size	size of trigger part	10 to 100 pages

Table 1: Workload Parameters Definitions and Values

Parameter	Description	Value
num_cpus	Number of CPUs	2 k CPUs
num_disks	Number of data disks	2 k Disks
k	Resource Unit	1
page_cpu	CPU time for accessing a page	10 millisecond
page_io_access	I/O time for accessing a page	35 milliseconds
log_disk_io	time for issuing a I/O log access	35 milliseconds
log_rec_io_w	I/O time for sequentially writing 1 page on log disk	1 milliseconds
commit_cpu	cpu time for executing a commit	10 milliseconds
abort_cpu	cpu time for executing an abort	10 milliseconds
restart_delay	restart delay of an aborted transaction	5 milliseconds
cpu_cc_request	cpu time for servicing one cc_request	1 millisecond

Table 2: Systems Parameters Definitions and Values

The physical queuing model (Figure 8b) is quite similar to the one used in [CM86]. The parameters *num_cpus* and *num_disks* specify the number of CPU servers and I/O servers. The requests to the CPU queue and the I/O queues are serviced FCFS (first come, first serve). Parameter *page_cpu* is the amount of CPU time for accessing a page. Parameter *Page_io_access* is the amount of I/O time associated with accessing a data page from the disk. We added to this model one separate I/O server dedicated to the log file. The parameter *log_disk_io* represents the fixed I/O time overhead associated with issuing the I/O. Parameter *log_rec_io_w* is the amount of I/O time associated with writing a log record on the Log disk in sequential order. Parameter *commit_cpu* is the amount of CPU time associated with executing the commit (releasing locks, and so on). Parameter *abort_cpu* is the amount of CPU time associated with executing the abort statement (executing undo operations, releasing locks and so on).

In order to simulate S2PL and EMV2PL we made some assumptions about their costs. First, we assumed that both algorithms incur a same cost noted *cpu_cc_request* to service the different concurrency control requests (setting locks, checking timestamp). This cost is charged at the points where each algorithm requires the associated action. Second, the scheme to maintain versions is the one used in [CM86] and inspired from the CCA algorithm of [CFL+82]. In this scheme, the I/O cost

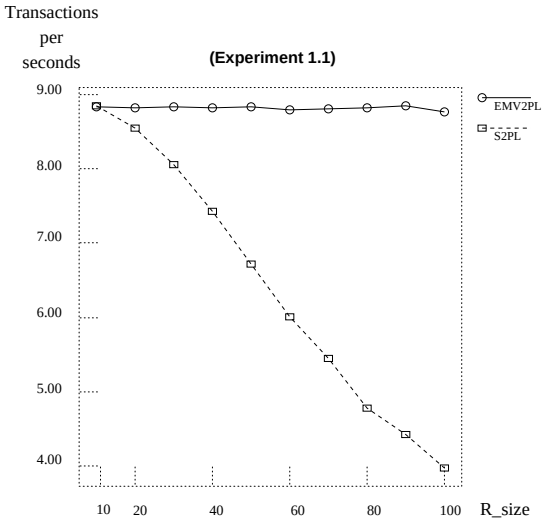

Figure 9: W transactions throughput

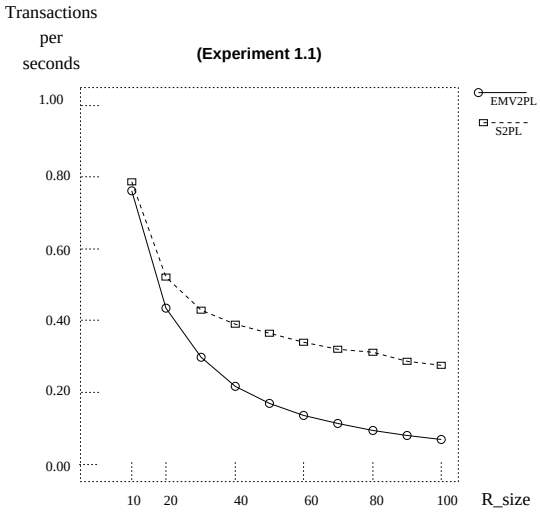

Figure 10: W|R transactions throughput

for accessing versions of an object varies if there is a pending uncommitted update for the object. If there is no pending update, accessing the i^{th} version of the object requires i disk accesses[13]. If there is a pending update, one disk access is required to access the first or the second version, two disk accesses for the third version, and so on. Finally, we assume, as explained in [CM86], that both single version and multiversion algorithms have similar recovery costs (i.e. log writes and version pool writes).

Each simulation consisted of 3 repetitions, each consisting of 1000 seconds of simulation time. These numbers were chosen in order to achieve more than 90 percent confidence intervals for our results. Table 2 summarizes the physical parameters and their values for the experiments.

6.2 Experiment 1

In this experiment, the database is divided into two tables $R1$ and $R2$, each containing 1500 pages. WR_frac is set to 20%. W|R transactions update $R1$ and execute triggers on $R2$ (as *purchase* transactions in Example 3) and W transactions read and write $R2$ (experiment 1.1), or $R1$ and $R2$ (experiment 1.2). This experiment evaluates the benefits of executing trigger parts under EMV2PL when $R2$ is a bottleneck for short concurrent W transactions (as *debit* transactions in Example 3). The variable in this experiment is the size of trigger parts (R_size) in order to vary the lock contention on $R2$.

Experiment 1.1: we first evaluate the performance of EMV2PL when W transactions update exclusively items of $R2$. The Figures 9 and 10 show respectively the W and W|R transactions throughput under S2PL or EMV2PL. This first set of results illustrates the well-known tradeoff of multiversion concurrency control compared to

[13] Versions are chained in reversal chronological order

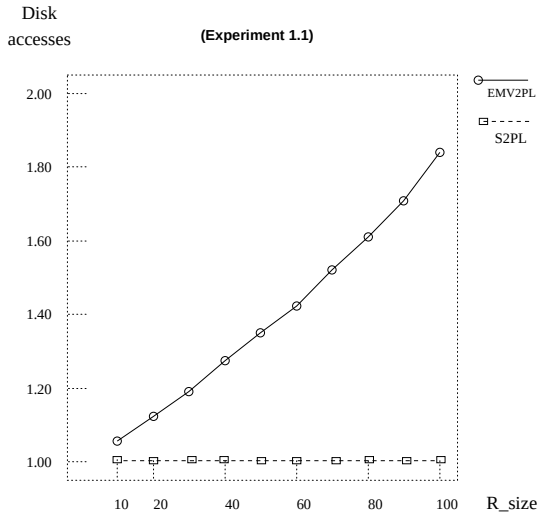

Figure 11: Average version accesses

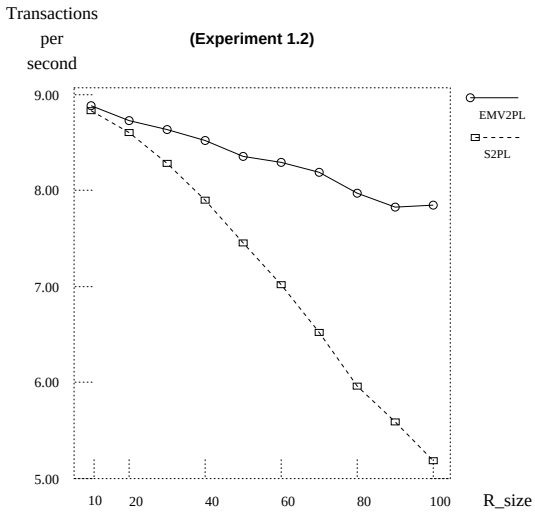

Figure 12: W transactions throughput

S2PL. S2PL provides good performance for long W|R transactions at the expense of short W transactions. The system resources are largely devoted to the execution of long W|R transactions while W transactions are blocked for a long time. In contrast, by completely eliminating blockings between W and W|R transactions, EMV2PL exhibits opposite results. The throughput of W transactions is unaffected by longer W|R transactions (they are not blocked anymore and compete again fairly for system resources). The throughput of W|R transactions is less since they perform more I/O for reading versions without gaining much from less blockings. Figure 11 shows the added I/O page accesses executed by W|R transactions in EMV2PL.

In experiment 1.1, the locks acquired on $R1$ by W|R transactions never block W transactions. This is an ideal case for EMV2PL. We now introduce conflicts between the program parts of W|R transactions and W transactions.

Experiment 1.2: W transactions pick randomly database items with an equal probability of accessing an item from $R1$ or $R2$. The Figure 12 shows the throughput of W transactions. We observe that the throughput of W transactions under EMV2PL decreases with increasing R_size. Indeed, lock conflicts on $R1$ are not eliminated by EMV2PL and mainly affect W transactions because of the long lock holding time of W|R transactions.

6.3 Experiment 2

In this experiment, we introduce conflicts between the program parts and the trigger parts of W|R transactions. Thus, the *check_read*() will now sometimes block the caller transactions. The point of this experiment is to evaluate the EMV2PL tradeoff when all types of conflicts occur. The following workload is used. Transactions randomly access pages of the whole database. The size of the trigger parts is held

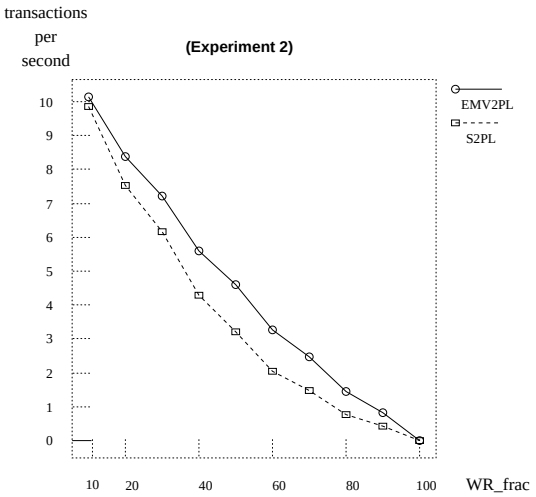

Figure 13: W transactions throughput

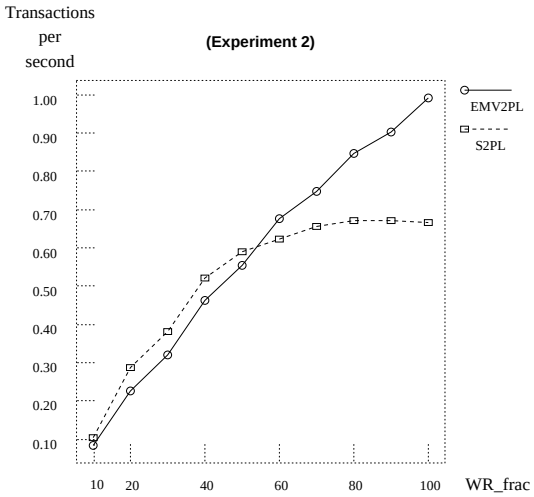

Figure 14: W|R transactions throughput

fixed (50 pages) and we vary WR_frac. This allows us to show how the benefits of EMV2PL and the storage cost vary with the transaction mix.

Figure 13 shows the throughput of W transactions. The decrease in throughput is due only to the variation of the transaction mix. Observe that EMV2PL always outperforms S2PL. The biggest benefits are obtained for values of WR_frac between 40% and 70% for which we observe increases of nearly 30% in throughput. Once again, this is because EMV2PL eliminates many conflicts which strongly affect W transactions in S2PL.

Figure 14 shows the throughput of W|R transactions. Surprisingly, EMV2PL outperforms S2PL when more than 55% of W|R transactions are running. This is explained as follows. When the mix consists mainly of W|R transactions, most conflicts occur between their trigger parts and program parts. These conflicts are significantly reduced with EMV2PL. For example when $WR\_frac = 1$, roughly half of these conflicts are eliminated[14]. EMV2PL also reduces deadlocks among W|R transactions (90 % reducing with $WR\_frac = 1$). Figure 15 shows that when WR_frac is high, W|R transactions do not suffer much from extra I/O overhead (less than 1.1 I/O access is needed per read operation with $WR\_frac > 50\%$). This is because most operations executed in such a mix are read operations and thus few versions are created. Finally, the Figure 16 shows the storage overhead of EMV2PL, measured as the average (relative) number of versions which may be needed to satisfy a read operation of some ongoing W|R transaction (i.e., at time t, it is the number of versions with timestamp smaller than the timestamp of the oldest W|R transaction). The greatest number is obtained with 80% W transactions, where the largest number of updates occur during the lifetime of the trigger parts of W|R transactions.

[14]Recall that *check_read*() blocks the caller only if it conflicts with a transaction with a smaller timestamp

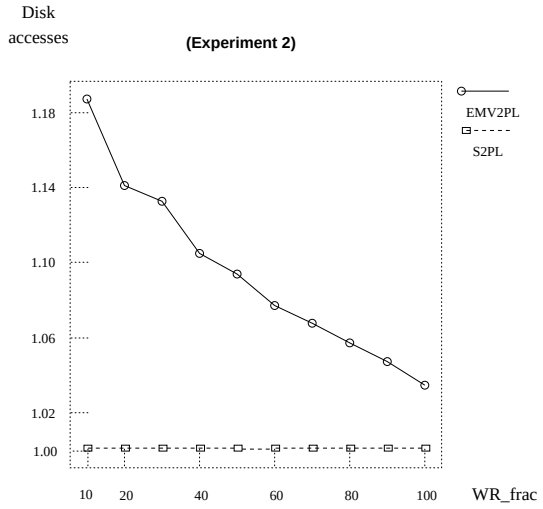

Figure 15: Average version accesses

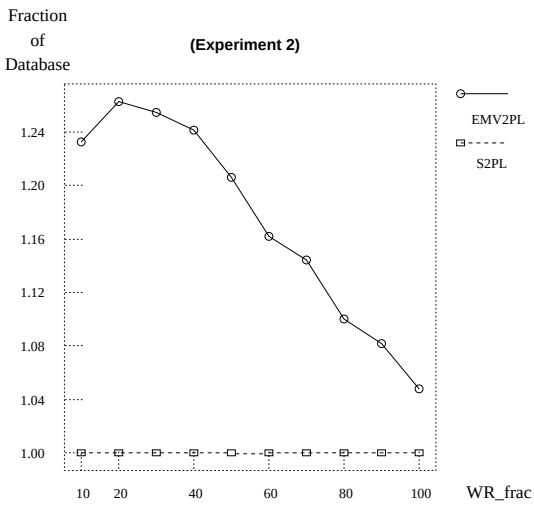

Figure 16: Relative storage overhead

6.4 Conclusions of the experiments

These experiments highlight situations where EMV2PL provides increased performance over S2PL. As expected, EMV2PL can significantly improve the performance of short W transactions if executed concurrently with long W|R transactions. This improvement is obtained at the expense of W|R transactions. This choice is a reasonable tradeoff since in most applications short transactions require the best response time. Furthermore, EMV2PL turns out to be also efficient under a mix mainly composed of long W|R transactions. The gain in performance in this case is achieved with a low storage overhead.

7 Related Work

A very few research papers have addressed the specific problems of optimizing active database transactions. A related idea, described in [DHL90], executes triggers in separate transactions from the triggering transaction. When a trigger is defined, a *decoupled* coupling mode can be chosen, indicating that the trigger is run in a separate transaction. This mode is subdivided into *dependent decoupled* where the separate transaction is not spawned unless the triggering transaction commits, and *independent decoupled*, where the separate transaction is spawned regardless of weather the triggering transaction commits. As shown in [CJL91], defining triggers in a decoupled mode may improve the throughput of concurrent transactions. However, the major problem for the design of triggers and the application programmer is verifying the correctness of a set of transactions that may trigger decoupled rules. Our solution takes a different approach since it preserves full isolation of all transactions.

An extensive literature addresses the problem of designing concurrency control algorithms that augment the performance of concurrent transactions. Our work directly builds on previous work on MV2PL protocols ([CFL+82], [BHG87], [AS89],

[AK91], [BC92], [MPL92]). We differ from these works by focusing on a particular class of transactions, namely Write-then-Read transactions, which are particularly interesting in an active database framework. In fact, every update transaction that executes alerters and integrity controls at the end of the transaction becomes a Write-then-Read transaction. In [BC92] and [MPL92], the authors propose techniques in which an update transaction does not systematically create a new version of its updated items. Hence, read-only transactions do not access versions which are the most up-to-date before their starting time. Instead, they all read a given older state which is periodically refreshed. The goal is to reduce the number of versions stored in the database ([BD92] and [MWC92] studied the impact of these techniques). This technique cannot be used in our protocol because the trigger part of W|R transactions must access the most up-to-date versions that precede their starting time. Otherwise, the trigger part of a W|R might read versions out-of date with respect to the one read by the program part, and a serialization fault would occur.

8 Conclusion

We have studied the problem of efficiently executing concurrent transactions that invoke the execution of deferred database triggers. We considered a restricted class of triggers, called RCA triggers, which represent general integrity constraints, and alerters. Our major contribution is to propose a slight extension of the MV2PL protocol, in which all read operations required by the execution of RCA triggers access appropriate versions and do not acquire locks. We proved the correctness of this protocol, and showed that it can be implemented by doing very few changes to an existing MV2PL implementation. Our simulation study shows that compared to strict two phase locking, our protocol significantly improves the performance of active database transactions by reducing transactions inter-blocking.

As future work, we first envision to improve our simulation model with a more sophisticated physical model. In particular, we wish to model the on-page caching of versions as proposed in [BC92]. Another direction of research is to select representative database transaction workloads and compare their performance when integrity controls are implemented by RCA triggers or when they are implemented by database procedures explicitly invoked by transactions. The intuitive idea is to balance the inherent overhead of using triggers with the advantage offered by our concurrency control protocol.

Acknowledgments

We are grateful to Anthony Tomasic for his detailed comments that enabled to improve this paper.

References

[ACL87] R. Agrawal, M.J. Carey, and M. Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Transactions on Computers and Systems*, 12(4):609–654, December 1987.

[ADW92] D. Adler, B. Dageville, and Kam-Fai Wong. *SIM : A C-based SIMultion Package*. ECRC-92-27i, 1992.

[AK91] D. Agrawal and V. Krishnaswamy. Using multiversion data for non-interfering execution of write-only transactions. *Proc. ACM SIGMOD International Conference on Management of Data, Denver, Colorado*, 20:98–107, May 1991.

[AS89] D. Agrawal and S. Sengupta. Modular synchronisation in multiversion databases: Version control and concurrency control. *Proc. ACM SIGMOD International Conference on Management of Data, Portland, Oregon*, pages 408–417, 1989.

[BC92] P. M. Bober and M. J. Carey. On mixing queries and transactions via multiversion locking. *Proc. International Conference on Data Engineering, Tempe, Arizona*, pages 535–545, February 1992.

[BD92] P. Bober and D.M. Dias. Storage cost tradeoffs for multiversion concurrency control. Technical Report RC 18367, IBM Research Division, T.J. Watson Research Center, July 1992.

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, 1987.

[Car83] M.J. Carey. *Modeling and Evaluation of Database Concurrency Control Algorithms*. PhD thesis, University of California, Berkeley, September 1983.

[CFL+82] A. Chan, S. Fox, W.K. Lin, A. Nori, and D.R. Ries. The implementation of an integrated concurrency control and recovery scheme. *Proc. ACM SIGMOD International Conference on Management of Data, Orlando, Florida*, pages 184–191, June 1982.

[CJL91] M. C. Carey, R. Jauhari, and M. Livny. On transaction boundaries in active databases : A performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, 3(3), September 1991.

[CM86] M. J. Carey and W. A. Muhanna. The performance of multiversion concurrency control algorithms. *ACM Transactions on Computers and Systems*, 4(4):338–378, November 1986.

[DHL90] U. Dayal, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. *Proc. ACM SIGMOD International Conference on Management of Data, Atlantic City, New Jersay*, pages 204–214, May 1990.

[GR93] J. Gray and A. Reuter. *Transaction Processing.* Morgan Kaufman, 1993.

[Mel93] J. Melton, editor. *(ISO/ANSI Working Draft) Database Language SQL3.* Number ANSI X3H2-90-412 and ISO DBL-YOK 003. February 1993.

[MPL92] C. Mohan, H. Pirahesh, and R. Lorie. Efficient and flexible methods for transient versionning of records to avoid locking by read-only transactions. *Proc. ACM SIGMOD International Conference on Management of Data, San Diego, California*, pages 124–133, June 1992.

[MWC92] A. Merchant, K.-L. Wu, and M.-S. Chen. Performance analysis of dynamic finite versioning for concurrent transaction and query-processing. *Proc. of ACM SIGMETRICS Int. Conf. on Measurement and Modeling of Computer Systems, Newport, Rhode island*, 20(1):103–114, June 1992.

[SWY93] H.-J. Schek, G. Weikum, and H. Ye. Towards a unified theory of concurrency control and recovery. *Proc. ACM Symposium on the Principles of Database Systems, Washington DC*, May 1993.

[Ull88] J.D. Ullman. *Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

[WCD95] J. Widom, S. Ceri, and U. Dayal. *Active Database System: Triggers and Rules for Advanced Database Processing.* Morgan-Kaufmann, San Fransisco, 1995. To appear.

Appendix 1

The code of the *check_read* function follows (see [GR93] for a precise description of the data structures used here). For simplicity, only the shared (S) and exclusive (X) mode are considered.

```
lock_reply check_read(lock_name name)
{
long bucket;                                        /*index of hash bucket*/
lock_head* head;                                    /*pointer to lock header block*/
lock_request* request;                              /*this lock request block*/
TransCB* me=MyTransCB();                            /*pointer to caller's transaction descriptor*/
bucket= lockhash(name);                             /*find hash chain*/
Xsem_get(&lock_hash[bucket].Xsem);                  /*get semaphore on it*/
head = lock_hash[bucket].chain;                     /*traverse hash chain*/
while((head !=NULL) && (head->name != name)         /*with this name*/
      {head = head->chain;};
/*lock already taken in X mode : */
if ((head != NULL) && (head->mode == X))
      { Xsem_get(&head->Xsem);                      /*acquire semaphore on lock header*/
        Xsem_give(&head_hash[bucket].Xsem);         /*release semaphore on lock chain*/
        request = head->queue                       /*the first request is the granted one*/
        if(request->tran == me)
            return(LOCK_OK);                        /*ok to write an item already X-locked by the caller*/
        else if (request->tran->tn < me->tn)        /*the read must be delayed ("critical read")*/
            { append(me->pid,head->pid_list);
              Xsem_give(&head->Xsem);               /*release semaphore on lock head*/
              wait();                               /*wait*/
              return(LOCK_OK);
            }
        else
            return(LOCK_OK);
        }
}
```

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399